Agenda

1. Overview
2. Basic syntax & structure
3. Compilation
4. Visibility & Lifetime
Agenda

5. Pointers & Arrays
6. Dynamic memory allocation
7. Composite data types
Not a Language Course!

- Resources:
  - K&R (The C Programming Language)
  - comp.lang.C FAQ (c-faq.com)
  - UNIX man pages (kernel.org/doc/man-pages/)
>man strlen

NAME
   strlen - find length of string

LIBRARY
   Standard C Library (libc, -lc)

SYNOPSIS
   #include <string.h>

       size_t
       strlen(const char *s);

DESCRIPTION
   The strlen() function computes the length of the string s.

RETURN VALUES
   The strlen() function returns the number of characters that precede the
   terminating NUL character.

SEE ALSO
   string(3)
§Overview
C is ...

- imperative
- statically typed
- weakly type checked
- procedural
- low level
<table>
<thead>
<tr>
<th>C</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedural</td>
<td>Object-oriented</td>
</tr>
<tr>
<td>Source-level portability</td>
<td>Compiled-code portability</td>
</tr>
<tr>
<td>Manual memory management</td>
<td>Garbage collected</td>
</tr>
<tr>
<td>Pointers reference addresses</td>
<td>Opaque memory references</td>
</tr>
<tr>
<td>Manual error code checking</td>
<td>Exception handling</td>
</tr>
<tr>
<td>Manual namespace partitioning</td>
<td>Namespaces with packages</td>
</tr>
<tr>
<td>Small, low-level libraries</td>
<td>Vast, high-level class libraries</td>
</tr>
</tbody>
</table>
§Basic syntax & structure
Primitive Types

- **char**: one byte integer (e.g., for ASCII)
- **int**: integer, *at least* 16 bits
- **float**: single precision floating point
- **double**: double precision floating point
Integer type prefixes

- signed (default), unsigned
  - same storage size, but sign bit on/off
- short, long
  - sizeof (short int) \(\geq\) 16 bits
  - sizeof (long int) \(\geq\) 32 bits
  - sizeof (long long int) \(\geq\) 64 bits
Recall C’s weak type-checking…

/* types are implicitly “converted” */
char c = 0x41424344;
short s = 0x10001000;
int i = 'A';
unsigned int u = -1;

printf("'\%c', \%d, \%X, \%X\n", c, s, i, u);

'D', 4096, 41, FFFFFFFF
Basic Operators

- Arithmetic: +, -, *, /, %, ++, --, &, |, ~
- Relational: <, >, <=, >=, ==, !=
- Logical: &&, ||, !
- Assignment: =, +=, *=, ...
- Conditional: bool ? true_exp : false_exp
True/False

- 0 = False
- **Everything else** = True
  - But *canonical* True = 1
Boolean Expressions

\(! (0)\) \rightarrow 1

0 || 2 \rightarrow 1

3 && 0 && 6 \rightarrow 0

\(! (1234)\) \rightarrow 0

!!(-1020) \rightarrow 1
Control Structures

- if-else
- switch-case
- while, for, do-while
- continue, break
Variables

- Must declare before use
- Declaration implicitly *allocates* storage for underlying data
- Note: not true in Java!
Functions

- C’s *top-level* modules
- Procedural language vs. OO: no classes!
Declaration vs. Definition

- Declaration (aka prototype): arg & ret type
- Definition: function body
- A function can be declared many times but only defined once
Declarations reside in *header* (.h) files, Definitions reside in *source* (.c) files

(Suggestions, not really requirements)
hashtable.h

```
#include "hashtable.h"

unsigned long hash(char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash;
}

hashtable_t *make_hashtable(unsigned long size) {
    hashtable_t *ht = malloc(sizeof(hashtable_t));
    ht->size = size;
    ht->buckets = calloc(sizeof(bucket_t *), size);
    return ht;
}
```

hashtable.c

```
#include "hashtable.h"

unsigned long hash(char *str) {
    unsigned long hash = 5381;
    int c;
    while ((c = *str++))
        hash = ((hash << 5) + hash) + c;
    return hash;
}

hashtable_t *make_hashtable(unsigned long size) {
    hashtable_t *ht = malloc(sizeof(hashtable_t));
    ht->size = size;
    ht->buckets = calloc(sizeof(bucket_t *), size);
    return ht;
}
...
# hashtable.h

```c
unsigned long hash(char *str);
hashtable_t *make_hashtable(unsigned long size);
void ht_put(hashtable_t *ht, char *key, void *val);
void *ht_get(hashtable_t *ht, char *key);
void ht_del(hashtable_t *ht, char *key);
void ht_iter(hashtable_t *ht, int (*f)(char *, void *));
void ht_rehash(hashtable_t *ht, unsigned long newsize);
int ht_max_chain_length(hashtable_t *ht);
void free_hashtable(hashtable_t *ht);
```

---

# main.c

```c
#include "hashtable.h"

int main(int argc, char *argv[]) {
    hashtable_t *ht;
    ht = make_hashtable(atoi(argv[1]));
    ...
    free_hashtable(ht);
    return 0;
}
```
§Compilation
```
#include <stdio.h>

int main () {
    printf("Hello world!\n");
    return 0;
}

$ gcc main.c -o prog
$ ./prog
Hello world!
```
```c
#include <stdio.h>
#include "greet.h"

void greet(char *name) {
    printf("Hello, %s\n", name);
}

int main() {
    greet("Michael");
    return 0;
}
```

```
greet.h

void greet(char *);

```
The diagram illustrates the process of compiling and linking multiple source files to create an executable. Here is a textual representation of the steps:

1. **Preprocessing**
   - `main.c`
   - `lib1.c`
   - `lib2.c`
2. **Compilation**
   - `main.c` -> `main.S`
   - `lib1.c` -> `lib1.S`
   - `lib2.c` -> `lib2.S`
3. **Assembly**
   - `main.S`
   - `lib1.S`
   - `lib2.S`
4. **Linking**
   - `main.o`
   - `lib1.o`
   - `lib2.o`
5. **Loading**
   - `a.out` (default)
“Preprocessing”

- preprocessor directives exist for:
  - text substitution
  - macros
  - conditional compilation
- directives start with ‘#’
$ gcc -E greet.c

```
void greet(char *);

void greet(char *name) {
    printf("Hello, %s\n", name);
}
```

stop and show source after preprocessing stage
#define msg "Hello world!\n"

int main () {
    printf(msg);
    return 0;
}

$ gcc -E hello.c

int main () {
    printf("Hello world!\n");
    return 0;
}
```c
#define PLUS1(x) (x+1)

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}

$ gcc -E plus1.c

int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```
```c
#define PLUS1(x) (x+1)

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

```c
#define PLUS1(x) x+1

int main () {
    int y;
    y = y * PLUS1(y);
    return 0;
}
```

$gcc -E plus1.c

```c
int main () {
    int y;
    y = y * (y+1);
    return 0;
}
```

$gcc -E plus1b.c

```c
int main () {
    int y;
    y = y * y+1;
    return 0;
}
```

no!

macros *blindly* manipulate text!
```c
int main () {
    int f0=0, f1=1, tmp;

    for (int i=0; i<20; i++) {
        #ifdef VERBOSE
            printf("Debugging: %d\n", f0);
        #endif
        tmp = f0;
        f0 = f1;
        f1 = tmp + f1;
    }
    return 0;
}
```
“Linking”

- Resolving symbolic references (e.g., variables, functions) to their definitions
  - e.g., by placing final target addresses in jump/call instructions
- Both static and dynamic linking are possible; the latter is performed at run-time
greet.h

```c
#include <stdio.h>
#include "greet.h"

void greet(char *);
```

greet.c

```c
#include <stdio.h>
#include "greet.h"

void greet(char *name) {
    printf("Hello, %s\n", name);
}
```

main.c

```c
#include "greet.h"

int main() {
    greet("Michael");
    return 0;
}
```

$ gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
$ objdump -d main.o
0000000000000000 <main>:
 0:   55                      push   %rbp
 1:   48 89 e5                mov    %rsp,%rbp
 4:   bf 00 00 00 00           mov    $0x0,%edi
 9:   e8 00 00 00 00           callq  e <main+0xe>
 e:   b8 00 00 00 00           mov    $0x0,%eax
13:   5d                      pop    %rbp
14:   c3                      retq

$ objdump -d greet.o
0000000000000000 <greet>:
 0:   55                      push   %rbp
 1:   48 89 e5                mov    %rsp,%rbp
 4:   bf 00 00 00 00           mov    $0x0,%edi
 9:   e8 00 00 00 00           callq  22 <greet+0x22>
 e:   b8 00 00 00 00           mov    $0x0,%eax
13:   c9                      leaveq
14:   c3                      retq
$ gcc -c greet.c     -o greet.o
$ gcc -c main.c      -o main.o
$ gcc greet.o main.o -o prog
$ ./prog
Hello, Michael
$ objdump -d prog

000000000004003f0 <printf@plt-0x10>:
  4003f0: ff 35 12 0c 20 00 pushq 0x200c12(%rip) # 601008 <GLOBAL_OFFSET_TABLE_+0x8>
  4003f6: ff 25 14 0c 20 00 jmpq *0x200c14(%rip) # 601010 <GLOBAL_OFFSET_TABLE_+0x10>
  4003fc: 0f 1f 40 00 nopl 0x0(%rax)

00000000000400400 <printf@plt>:
  400400: ff 25 12 0c 20 00 jmpq *0x200c12(%rip) # 601018 <GLOBAL_OFFSET_TABLE_+0x18>
  400406: 68 00 00 00 00 pushq $0x0
  40040b: e9 e0 ff ff ff jmpq 4003f0 <_init+0x28>

00000000000400526 <main>:
  400526: 55 push %rbp
  400527: 48 89 e5 mov %rsp,%rbp
  40052a: bf e4 05 40 00 mov $0x4005e4,%edi
  40052f: e8 07 00 00 00 callq 40053b <greet>
  400534: b8 00 00 00 00 mov $0x0,%eax
  400539: 5d pop %rbp
  40053a: c3 retq

0000000000040053b <greet>:
  40053b: 55 push %rbp
  40053c: 48 89 e5 mov %rsp,%rbp
  40053f: 48 83 ec 10 sub $0x10,%rsp
  400543: 48 89 7d f8 mov %rdi,-0x8(%rbp)
  400547: 48 8b 45 f8 mov -0x8(%rbp),%rax
  40054b: 48 89 c6 mov %rax,%rsi
  40054e: bf ec 05 40 00 mov $0x4005ec,%edi
  400553: b8 00 00 00 00 mov $0x0,%eax
  400558: e9 a3 fe ff ff callq 400400 <printf@plt>
  40055d: 90 nopl
  40055e: c9 leaveq
  40055f: c3 retq
“Linking”

- I.e., the linker allows us to create large, multi-file programs with complex variable/function cross-referencing

- Pre-compiled libraries can be “linked in” (statically or dynamically) without rebuilding from source
“Linking”

- But, we don’t always want to allow linking a call to a definition!

- e.g., to hide implementations and build selective public APIs
§ Visibility & Lifetime
Visibility: *where* can a symbol (var/fn) be seen from, and how do we refer to it?

Lifetime: *how long* does allocated storage space (e.g., for a var) remain useable?
$ gcc -Wall -o demo sum.c main.c
sum.c: In function `sumWithI':
  sum.c:2: error: `I' undeclared (first use in this function)
main.c: In function `main':
  main.c:6: warning: implicit declaration of function `sumWithI'

int sumWithI(int x, int y) {
  return x + y + I;
}

#include <stdio.h>

int I = 10;

int main() {
  printf("%d\n", sumWithI(1, 2));
  return 0;
}
```c
#include <stdio.h>

int sumWithI(int x, int y) {
    int I;
    return x + y + I;
}

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
-1073743741
```
problem: variable *declaration* & *definition* are implicitly tied together

note: definition = *storage allocation* + possible *initialization*
`extern` keyword allows for declaration *sans definition*
int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}

#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}

$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
… and now global variables are visible from everywhere.

Good/Bad?
static keyword lets us limit the *visibility* of things
```c
#include <stdio.h>

int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
  "_I", referenced from:
    _sumWithI in ccmvi0RF.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
sum.c

```c
static int sumWithI(int x, int y) {
    extern int I;
    return x + y + I;
}
```

main.c

```c
#include <stdio.h>

int sumWithI(int, int);

int I = 10;

int main() {
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

$ gcc -Wall -o demo sum.c main.c
Undefined symbols:
"_sumWithI", referenced from:
    _main in cc9LhUBP.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
static also forces the *lifetime* of variables to be equivalent to *global* (i.e., stored in static memory vs. stack)
```c
#include <stdio.h>

int sumWithI(int, int); 

int main() {
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    printf("%d\n", sumWithI(1, 2));
    return 0;
}
```

```
$ gcc -Wall -o demo sum.c main.c
$ ./demo
13
14
15
```
§ Pointers
(don’t panic!)
a *pointer* is a variable declared to store a *memory address*
Q: by examining a variable’s contents, can we tell if the variable is a pointer?

e.g., \(0x0040B100\)
No!

- a pointer is designated by its *static* (declared) *type*, not its contents
A pointer declaration also tells us the type of data to which it should point.
declaration syntax: type *var_name
int *ip

char *cp;

struct student *sp;
Important pointer-related operators:

& : address-of

* : dereference *(not the same as the * used for declarations!!!)*
int i = 5;  /* i is an int containing 5 */
int *p;    /* p is a pointer to an int */
p = &i;    /* store the address of i in p */

int j;    /* j is an uninitialized int */
j = *p;    /* store the value p points to into j*/
```c
int main() {
    int i, j, *p, *q;

    i = 10;
    p = &j;
    q = p;
    *q = i;
    *p = *q * 2;
    printf("i=%d, j=%d, *p=%d, *q=%d\n", i, j, *p, *q);
    return 0;
}
```

```
$ gcc pointers.c
$ ./a.out
i=10, j=20, *p=20, *q=20
```
```c
int i, j, *p, *q;
i = 10;
p = &j;
qu = p;

*i = i;
*p = *q * 2;
```
int main() {
    int i, j, *p, *q;
    i = 10;
    p = &j;
    q = p;
    *q = i;
    *p = *q * 2;
    return 0;
}

main:
    pushq %rbp
    movq %rsp, %rbp
    movl $10, -4(%rbp)
    leaq -28(%rbp), %rax
    movq %rax, -16(%rbp)
    movq -16(%rbp), %rax
    movq %rax, -24(%rbp)
    movq -24(%rbp), %rax
    movl -4(%rbp), %edx
    movl %edx, (%rax)
    movq -24(%rbp), %rax
    movl (%rax), %eax
    leal (%rax,%rax), %edx
    movq -16(%rbp), %rax
    movl %edx, (%rax)
    movl $0, %eax
    popq %rbp
    ret

(via Compiler Explorer: https://godbolt.org)
why have pointers?
```c
int main() {
    int a = 5, b = 10;
    swap(a, b);
    /* want a == 10, b == 5 */
    ...
}

void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```
```c
int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    /* want a == 10, b == 5 */
    ...
}

void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}
```
pointers enable *action at a distance*
```c
void bar(int *p) {
    *p = ...; /* change some remote var! */
}

void bat(int *p) {
    bar(p);
}

void baz(int *p) {
    bat(p);
}

int main() {
    int i;
    baz(&i);
    return 0;
}
```
action at a distance is an *anti-pattern*

i.e., an oft used but typically crappy programming solution
back to swap

```c
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a = 5, b = 10;
    swap(&a, &b);
    /* want a == 10, b == 5 */
    ...
}
```
... for swapping pointers?

```c
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c, *d;
    c = &a;
    d = &b;

    swap(c, d);
    /* want c to point to b, d to a */
...
}
```
void swap(int *p, int *q) {
    int tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c = &a, *d = &b;

    swap(&c, &d);
    /* want c to point to b, d to a */
}
void swapp(int **p, int **q) {
    int *tmp = *p;
    *p = *q;
    *q = tmp;
}

int main() {
    int a, b, *c = &a, *d = &b;
    swapp(&c, &d);
    /* want c to point to b, d to a */
}

(int **) declares a pointer to a pointer to an int
Uninitialized pointers

- are like all other uninitialized variables
  - i.e., contain garbage
- dereferencing garbage ...
  - if lucky → crash
  - if unlucky → ???
“Null” pointers

- never returned by \& operator
- safe to use as sentinel value
- written as 0 in pointer context
  - for convenience, \#define'd as NULL
“Null” pointers

```c
int main() {
    int i = 0;
    int *p = NULL;

    ...

    if (p) {
        /* (likely) safe to deref p */
    }
}
```
§ Arrays
contiguous, indexed region of memory
Declaration: `type arr_name[size]`
- remember, declaration also allocates storage!
int i_arr[10];          /* array of 10 ints */
char c_arr[80];         /* array of 80 chars */
char td_arr[24][80];    /* 2-D array, 24 rows x 80 cols */
int *ip_arr[10];        /* array of 10 pointers to ints */

/* dimension can be inferred if initialized when declaring */
short grades[] = { 75, 90, 85, 100 };

/* can only omit first dim, as partial initialization is ok */
int sparse[][][10] = {
    { 5, 3, 2 },
    { 8, 10 },
    { 2 }
};

/* if partially initialized, remaining components are 0 */
int zeros[1000] = { 0 };

/* can also use designated initializers for specific indices*/
int nifty[100] = {
    [0] = 0,
    [99] = 1000,
    [49] = 250
};
In C, arrays contain *no metadata* i.e., *no implicit size, no bounds checking*
```c
int main() {
    int i, arr[10];

    for (i=0; i<100; i++) {
        arr[i] = 0;
    }
    printf("Done\n");

    return 0;
}
```

```
$ gcc arr.c
$ ./a.out

(runs forever ... no output)
```
int main() {
    int arr[10], i;

    for (i=0; i<100; i++) {
        arr[i] = 0;
    }
    printf("Done\n");

    return 0;
}

$ gcc arr.c
$ ./a.out
Done
[1]    10287 segmentation fault  ./a.out
$
this is the basis of *buffer overrun* attacks!

what can you do with stack manipulation?

- code injection
- return redirection
- et al
direct access to memory can be *dangerous*!
pointers ❤ arrays

- an array name is bound to the address of its first element
  - i.e., array name is a \textit{const pointer}

- conversely, a pointer can be used as though it were an array name
int *pa;
int arr[5];

pa = &arr[0];  /* <=> */  pa = arr;

arr[i];  /* <=> */  pa[i];

*arr;  /* <=> */  *pa;

int i;

pa = &i;  /* ok */

arr = &i;  /* not possible! */
§ Pointer Arithmetic
follows naturally from allowing array subscript notation on pointers
int arr[100];

int *pa = arr;

pa[10] = 0;    /* set tenth element */

/* so it follows ... */

*(pa + 10) = 0; /* set tenth element */

/* surprising! "adding" to a pointer accounts for element size -- does not blindly increment address */
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

pa[10] = 0;

printf("%X\n", arr[10]);

$ ./a.out
DEADBEEF
```c
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

int offset = 10 * sizeof (int);

*(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
DEADBE00
```

**sizeof**: an operator to get the size in bytes - can be applied to a datum or type
```c
int arr[100];
arr[10] = 0xDEADBEEF;

char *pa = (char *)arr;

int offset = 10 * sizeof(int);

*(int *)(pa + offset) = 0;

printf("%X\n", arr[10]);
```

```
$ ./a.out
0
```
takeaway:

- pointer arithmetic makes use of pointee data types to compute byte offsets
strings are just $\theta$ terminated char arrays
```c
char str[] = "hello!";
char *p = "hi";
char tarr[][5] = {"max", "of", "four"};
char *sarr[] = {"variable", "length", "strings"};
```
/* printing a string (painfully) */

int i;
char *str = "hello world!";
for (i = 0; str[i] != 0; i++) {
    printf("%c", str[i]);
}

/* or just */

printf("%s", str);
/* Beware: */

```c
int main() {
    char *str = "hello world!";
    str[12] = 10;
    printf("%s", str);
    return 0;
}
```

```
$ ./a.out
[1] 22432 segmentation fault (core dumped) ./a.out
```
/* the fleshed out "main" with command-line args */

int main(int argc, char *argv[]) {
    int i;
    for (i=0; i<argc; i++) {
        printf("%s", argv[i]);
        printf("%s", ((i < argc-1)? "," : "\n") );
    }
    return 0;
}

$ ./a.out testing one two three
./a.out, testing, one, two, three
§ Dynamic Memory Allocation
**dynamic** vs. **static** (lifetime = forever)

vs. **local** (lifetime = LIFO)
C requires explicit memory management
- must request & free memory manually
- if forget to free → memory leak
vs., e.g., Java, which has *implicit* memory management via *garbage collection*

- allocate (via `new`) & forget!
basic C “malloc” API (in stdlib.h):

- malloc
- realloc
- free
malloc lib is *type agnostic*

i.e., it doesn’t care what data types we store in requested memory
need a “generic” / type-less pointer:

(\texttt{void *})
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);

all sizes are in bytes;
all ptrs are from previous malloc requests
int i, j, k=1;
int *jagged_arr[5]; /* array of 5 pointers to int */
for (i=0; i<5; i++) {
    jagged_arr[i] = malloc(sizeof(int) * k);
    for (j=0; j<k; j++) {
        jagged_arr[i][j] = k;
    }
    k += 1;
}

/* use jagged_arr ... */
for (i=0; i<5; i++) {
    free(jagged_arr[i]);
}
§Composite Data Types
≈ objects in OOP
C structs create user defined types, based on primitives (and/or other UDTs)
/* type definition */
struct point {
    int x;
    int y;
}; /* the end ';' is required */

/* point declaration (& alloc!) */
struct point pt;

/* pointer to a point */
struct point *pp;

/* combined definition & decls */
struct point {
    int x;
    int y;
} pt, *pp;
component access: dot (\'\.'\) operator

```c
struct point {
    int x;
    int y;
} pt, *pp;

int main() {
    pt.x = 10;
    pt.y = -5;

    struct point pt2 = { .x = 8, .y = 13 }; /* decl & init */

    pp = &pt;

    (*pp).x = 351; /* comp. access via pointer */

    ...
}
```
$ gcc point.c
... error: request for member ‘x’ in something not a structure or union

(*pp).x = 351;  \times \quad *pp.x = 351;

‘.’ has higher precedence than ‘*’
But \((\ast pp).x\) is painful

So we have the ‘\(-\rightarrow\)’ operator
- component access via pointer

```c
struct point {
    int x;
    int y;
} pt, *pp;

int main() {
    pp = &pt;
    pp->x = 10;
    pp->y = -5;

    ...
}
```
/* Dynamically allocating structs: */

struct point *parr1 = malloc(N * sizeof(struct point));
for (i=0; i<N; i++) {
    parr1[i].x = parr1[i].y = 0;
}

/* or, equivalently, with calloc (which zero-inits) */
struct point *parr2 = calloc(N, sizeof(struct point));

/* do stuff with parr1, parr2 ... */

free(parr1);
free(parr2);
In C all args are \textit{pass-by-value}!

```c
void foo(struct point pt) {
    pt.x = pt.y = 10;
}

int main() {
    struct point mypt = { .x = 5, .y = 15 };
    foo(mypt);
    printf("(%d, %d)\n", mypt.x, mypt.y);
    return 0;
}
```

(5, 15)
/* self referential struct */
struct ll_node {
    char *data;
    struct ll_node next;
};

$ gcc ll.c
ll.c:4: error: field ‘next’ has incomplete type

problem: compiler can’t compute size of next — depends on size of ll_node, which depends on size of next, etc.
/* self referential struct */
struct ll_node {
    char *data;
    struct ll_node *next; /* need a pointer! */
};

struct ll_node *prepend(char *data, struct ll_node *next) {
    struct ll_node *n = malloc(sizeof(struct ll_node));
    n->data = data;
    n->next = next;
    return n;
}

void free_llist(struct ll_node *head) {
    struct ll_node *p=head, *q;
    while (p) {
        q = p->next;
        free(p);
        p = q;
    }
}
main() {
    struct ll_node *head = 0;

    head = prepend("reverse.", head);
    head = prepend("in", head);
    head = prepend("display", head);
    head = prepend("will", head);
    head = prepend("These", head);

    struct ll_node *p;
    for (p=head; p; p=p->next) {
        printf("%s ", p->data);
    }
    printf("\n");

    free_llist(head);
}
very handy tool for detecting/debugging memory leaks: **valgrind**
main() {
    struct ll_node *head = 0;

    head = prepend("reverse.", head);
    ...

    // free_llist(head);
}
void free_llist(struct ll_node *head) {
    struct ll_node *p=head, *q;
    while (p) {
        //q = p->next;
        free(p);
        p = p->next;
    }
}

main() {  
    struct ll_node *head = 0;
    head = prepend("reverse.", head);
    ...
    free_llist(head);
}

# valgrind --leak-check=full ./12c-dma
==322== Invalid read of size 8
==322==  at 0x109212: free_llist (12c-dma.c:31)
==322==   Address 0x4a47188 is 8 bytes inside a block of size 16 free'd
==322==    by 0x10920D: free_llist (12c-dma.c:30)
==322==   Block was alloc'd at
==322==    by 0x1091C6: prepend (12c-dma.c:20)
==322==
==322== HEAP SUMMARY:
==322== in use at exit: 0 bytes in 0 blocks
==322== total heap usage: 6 allocs, 6 frees, 1,104 bytes allocated
==322==
==322== All heap blocks were freed -- no leaks are possible
</C_Primer>